Constraint Programming for Context Comprehension

Henning Christiansen

Abstract A close similarity is demonstrated between context comprehension, such as discourse analysis, and constraint programming. The constraint store takes the role of a growing knowledge base learned throughout the discourse, and a suitable constraint solver does the job of incorporating new pieces of knowledge. The language of Constraint Handling Rules, CHR, is suggested for defining constraint solvers that reflect "world knowledge" for the given domain, and driver algorithms may be expressed in Prolog or additional rules of CHR. It is argued that this way of doing context comprehension is an instance of abductive reasoning. The approach fits with possible worlds semantics that allows both standard first-order and non-monotonic semantics.

1 Introduction

There is a striking similarity between constraint logic programming and context comprehension. In both cases, a given structure is traversed, and bits of information are accumulated in a growing knowledge base. Constraint programming usually deals with structures that encode complex mathematical problems to be solved, and context comprehension with observed phenomena such as a spoken or written discourse, streaming sensor data from an industrial plant or a transport network, etc. As more knowledge is accumulated, the possible solution space in the constraint logic programming case and the set of possible worlds that represents context, will decrease, i.e., become more and more specific. For each step of computation in either paradigm, the adding of a new piece of information may involve a normalization and consistency check with respect to previous knowledge, and in case of an inconsistency, the overall process may change its control path, e.g., by backtracking.

Henning Christiansen
Roskilde University, P.O.Box 260, DK-4000 Roskilde, Denmark e-mail: henning@ruc.dk

Constraint logic programming is typically based on a hardcoded set of constraint predicates with a fixed semantics, tailored for a specific class of mathematical problems (as described, e.g., by [4, 25]) and may not be of much use for context comprehension from, say, linguistic utterances. However, a declarative programming language for defining constraint solvers such as Constraint Handling Rules [19, 20], for short CHR, changes the picture. With this "white box" approach to constraint solving, it becomes feasible to define constraint domains and solvers for specific knowledge representations, especially tailored for representing context.

In this chapter, we summarize and exemplify how a combination of CHR and Prolog, the latter used for driver algorithms such as linguistic parsers or other structural scanners, can be used to specify and implement different tasks of context comprehension. However, the overall architectural principles are not tied to these specific programming languages, but can be integrated with, say, more advanced systems for linguistic parsing, or – instead of using constraint solvers written in CHR – other knowledge management tools and representation formalisms may be used.

By context, we refer to the set of circumstances in which a particular phenomenon ϕ is observed and to which ϕ may owe its existence. For example, it is difficult to observe a hole in a doughnut (an example of a ϕ) without its context, the doughnut. For a given discourse, its context may be represented as a knowledge base about the particular circumstances explained. References to context in the discourse may be given explicitly in factual form or indirectly assumed. For example *"He likes Mary"*, presupposes that there is a male character in the context as well as a female one named Mary (as "Mary" is usually a female name) plus a fact about the relationship between those to characters. Context may be static when the given discourse describes universal properties (such as a math textbook) or some state of affairs in a stable period of time; in this case, the dynamic aspects are limited to the sequential traversal of the text (i.e., the reader's mental time) during which more and more information is recognized. A discourse may involve further dynamic aspect. It may describe developments in past times as in a history textbook, or the time of each utterance may be essential as in a psychological drama or a running commentary on a football match. In such cases context may be seen as a knowledge base of timestamped facts (as for the history book), or we may be interested in maintaining a representation of the a current "now" context, which means that each new observation may give rise to a revision of previous knowledge. The framework and analysis methods introduced below can handle these different modes of context.

The focus on context comprehension, rather than a purely compositional analysis of the observed phenomena, displays a similarity with abductive reasoning (in the sense of C.S. Pierce; see [3] for a modern exposition): the task is to figure out a feasible context in which a given phenomenon may be observed, or in logical terms, that the observation must be deductively derivable from the inherent background knowledge and the context to be identified. As spelled out in more details in section 4.1, it can be shown that Prolog with CHR is an instance – and efficient implementation – of so-called Abductive Logic Programming, and in this way our approach to context comprehension confirms the metaphor of "Interpretation as abduction" introduced in the often cited paper [24].

In section 2, we define constraints and constraint solving in a precise way with possible worlds semantics, and section 3 introduces CHR and demonstrates how it can be used for defining semantics and solvers. Section 4 describes driver algorithms in Prolog, elaborates on the relation to abduction and shows an example of Prolog's grammar notation used with CHR for context comprehension from. Finally, section 5 discusses related work and background sources, and section 6 provides a summary and conclusions.

2 Constraints and Constraint Solving

Traditional constraint programming is concerned with finding values for variables that will satisfy logically specified conditions about those variables. As an informal example, let us consider constraint predicates "$\in \mathscr{N}$" indicating a natural number, and $>$ for the usual ordering relation. The conjunction

$$x \in \mathscr{N} \wedge x > 5 \wedge x < 7 \tag{1}$$

is a constraint problem in x, and it has as solution $x = 6$. This short formula is in solved or normalized form: it is accepted as a standard format for delivering solutions. A solved form may express intensional answers, providing a finite representation of a perhaps infinite set of solutions. For example, the constraint problem $x \in \mathscr{N} \wedge x > 5 \wedge x > 7 \wedge x \neq 3$ may have the solved form $x \in \mathscr{N} \wedge x > 7$, that represents the infinite set of solutions $\{x = 8, x = 9, \ldots\}$.

When constraints are used for representing context, we also pay attention to variable-free constraints as having important content themselves, being statements about the world. So if "*raining*" is a constraint predicate, we may consider the formula *raining* as being true in any world where it is actually raining, and false in all other worlds. It can be read as an intensional representation of an infinite set of worlds in which it is definitely raining, some in which the sky is densely covered with clouds and yet others with a bit of sunshine and maybe a rainbow. We may still be interested in using variables, so for example "$\textit{tall-person}(x)$" may designate a collection of contexts or worlds in which there exists a tall person, although we do not know anything more specific about this person.

What we define below as a standard semantics complies with first-order logic, but we present our definitions in a more general way that also allows for non-monotonicity. A formula or other object is *ground* if it contains no variables; substitutions and grounding substitutions are defined in the usual way.

Definition 1. A *constraint framework* $F = \langle \mathscr{C}, \mathscr{W}, W \rangle$ consists of a set of *constraint predicates* $\mathscr{C}$, a set of *possible worlds* $\mathscr{W}$ and a *semantic function* W. The constraint predicates are assumed (without mention) to include the zero-ary predicates *true* and *false*; an atom whose predicate is in $\mathscr{C}$ is referred to as a *constraint*. A *constraint store* is a finite set of constraints, which may be written as a conjunction $c_1 \wedge \cdots \wedge c_n$; the empty store is identified with *true*, and *false* will be used also as a prototypical inconsistent constraint store (below). The symbol $\mathscr{S}$ (with subscript F understood) refers to the set of all constraint stores and $\overline{\mathscr{S}}$ specifically to the ground ones. The semantic function is given by a mapping $W\colon \overline{\mathscr{S}} \to 2^{\mathscr{W}}$ with

- $W(\mathit{true}) = \mathscr{W}$,
- $W(s_1 \wedge \mathit{true} \wedge s_2) = W(s_1 \wedge s_2)$ and
- $W(\mathit{false}) = W(\cdots \wedge \mathit{false} \wedge \cdots) = \emptyset$.

The semantics is generalized to nonground constraint stores as follows.

$$W(s) =_{\text{def}} \bigcup_{\sigma \text{ a grounding subst. for } s} W(s\sigma) \tag{2}$$

In case $w \in W(s)$, we may write $w \models s$ (F understood). A constraint store s is *consistent* if there is a world w such that $w \models s$; any other constraint store is *inconsistent*.

Notice that formula (2) corresponds to an implicit existential quantification of a constraint store at the outermost level. In practice, not all conjunctions of constraints will appear as constraint stores in which case we may leave their semantics unspecified or implicitly $\emptyset$.

Definition 2. A *standard semantics* W is one in which $W(s_1 \wedge s_2) = W(s_1) \cap W(s_2)$ for any ground constraint stores s_1, s_2.

Standard semantics includes first-order logical theories, in which $\mathscr{W}$ may be identified with a class of first-order models. Non-standard semantics include non-monotonic semantics in which certain constraints can be understood as update instructions, e.g., by deleting or replacing information.

Example 1. We consider a constraint framework whose constraints are of the form $in(i,\ell)$, where i is an individual in $\{1,2,3\}$ and ℓ a location in $\{a,b,c\}$. A possible world is any set of three ground atoms of the form $\{in(1,\ell_1), in(2,\ell_2), in(3,\ell_3)\}$, $\ell_1,\ell_2,\ell_3 \in \{a,b,c\}$, i.e., each world represents a scene in which each individual is placed in some location. We assume a standard semantics in which a store s is mapped into the set of worlds in which s holds, more precisely $W(s) = \{w \mid s \subseteq w\}$. This yields, for example,

- $W(in(1,a) \wedge in(2,b)) = \{\,\{in(1,a), in(2,b), in(3,\ell)\} \mid \ell \in \{a,b,c\}\}$,
- $W(in(1,a) \wedge in(1,b)) = W(in(1,a)) \cap W(in(1,b)) = \emptyset$.

In other words, the semantics has the inherent limitation that an individual can only be in one location at a time, but a given location may host any number of individuals, from 0 to 3.

In the following we give a definition of constraint solving that merges one constraint at a time into a developing, normalized store, rather that crunching a huge bunch of constraints in one go. There are several reasons for this.

- It fits well with discourse and text analysis, in which utterances arrive sequentially and the decomposition and translation into constraints of each utterance can exploit the context learned so far (very much like we humans do).

- It corresponds to the way a Prolog interpreter enhanced with constraint solving typically works: A proof is built in Prolog's traditional, recursive and goal-directed manner, but whenever a constraint is encountered, the constraint solver incorporates it into the growing constraint store, and if this succeeds, Prolog continues in the usual way.
- The sequential order of processing constraints is essential for implementing non-standard semantics.

Definition 3. Given a constraint framework $F = \langle \mathscr{C}, \mathscr{W}, W \rangle$ a set of *normalized* constraint stores is assumed, which includes particularly *true* and *false*. A normalized constraint store s' is a *normalized form of s* whenever, for any substitution σ for variables of s' and s, that $W(s'\sigma) = W(s\sigma)$.[1]

A *constraint solver* for a semantics W is a mapping, denoted $s_1, c \vdash s_2$, from a normalized constraint store s_1 and a new constraint c into a new normalized constraint store s_2 where s_2 is a normalized form of $s_1 \wedge c$. A constraint solver is *sound* if the normalized form of any inconsistent constraint store is *false* (meaning that the constraint solver recognizes inconsistency immediately, signaling this by the result *false*).

We introduce a convenient notation. When a semantics and constraint solver is given, the state s_n following from a sequence of insertions of constraints $c_1, \ldots, c_n$ into an initial store s_0 is denoted $[s_0, c_1, \ldots, c_n]$. More precisely,

$$\begin{aligned} s_0, c_1 &\vdash s_1 \\ s_1, c_2 &\vdash s_2 \\ &\vdots \\ s_{n-1}, c_n &\vdash s_n =_{\text{def}} [s_0, c_1, \ldots, c_n] \end{aligned}$$

When $s_0 = true$, it may be left out.

In the special case of a standard semantics, permuting the constraints may yield syntactically different states which will be equivalent according to the given semantics, i.e., they represent the same set of possible worlds.

Example 2. A constraint solver for the constraint framework of Example 1 can be described as follows.

$$\begin{array}{ll} s, in(i,\ell) \vdash s \cup \{in(i,\ell)\} & \text{whenever there is no } in(i,\ell') \in s \text{ with } \ell \neq \ell', \\ s, in(i,\ell) \vdash false & \text{otherwise.} \end{array}$$

[1] The mentioning of the substitution σ in Definition 3 is necessary in order to preserve the identity of variables in the store and its normalized version.

3 Defining Semantics and Constraint Solvers with CHR

Often the possible worlds semantics will be implicit in the definition of the actual constraints and constraint solver. A declarative programming language such as Constraint Handling Rules intended for defining constraint solvers is interesting here. Claiming a programming language to be "declarative" means that its programs can be read as a concise specification of what the program is supposed to accomplish without unnecessary computational details. In the following, we give first a brief introduction to Constraint Handling Rules and show, then, how it can be used for defining constraint solvers that may comply with a standard or a non-standard semantics.

3.1 Constraint Handling Rules: A Brief Introduction

Constraint Handling Rules (CHR) is an extension to the logic programming language Prolog that adds mechanisms for forward-chaining reasoning to complement Prolog's standard backward-chaining, goal directed reasoning. CHR is now part of several major Prolog systems, including SWI and Sicstus. Here we give only a very brief introduction; a comprehensive account on CHR and its applications can be found in the book [20].

CHR was originally intended as a declarative language for writing constraint solvers for standard constraint domains concerned with numbers, arithmetic, equations and the like. Later it has turned out that CHR is suited for automated reasoning in general as documented by the vast literature on applications of CHR, also summarized in [20]. As shown in, e.g., [11] and the present paper, the combination of CHR and Prolog is a powerful paradigm for implementing a variety of forms of reasoning and knowledge representations.

A CHR program consists of declarations of constraint predicates and rewriting rules over constraint stores. A simplified explanation of CHR's procedural semantics is that whenever a new constraint as called, it is included in the constraint store and the rules of the current program apply as long as possible. CHR has three sorts of rules of the following forms.

Simplification rules:	$h_1,\dots,h_n$ <=> $\mathit{Guard} \mid b_1,\dots,b_m$
Propagation rules:	$h_1,\dots,h_n$ ==> $\mathit{Guard} \mid b_1,\dots,b_m$
Simpagation rules:	$h_1,\dots,h_k \setminus h_{k+1},\dots h_n$ <=> $\mathit{Guard} \mid b_1,\dots,b_m$

The *h*'s are head constraints and *b*'s body constraints, and *Guard* is a guard condition (typically testing values of variables found in the head). A rule can be applied when its head constraints are matched simultaneously by constraints in the store and the guard is satisfied. For a simplification rule, the matched constraints are removed and the suitably instantiated versions of the body constraints are added. The other rules execute in a similar way, except that for propagation, the head constraints stay in the store, and for simpagation, only those following the backslash are removed. Prolog calls inside the body are executed in the usual way.

CHR has a logical semantics based on reading a simplification as a bi-implication, a propagation as an implication, and finally considering a simpagation $H_1 \setminus H_2 \texttt{<=>} G \mid B$ as equivalent with the simplification $H_1, H_2 \texttt{<=>} G \mid H_1, B$ (although it is executed in a different way). It is possible to write CHR programs that are inconsistent according to this semantics and non-termination can be an issue as well.

It can be shown, see [20], that if a program is terminating and confluent (roughly: the final result is independent of the order in which rules are applied), then it is consistent. Procedurally, when a new constraint arrives, the interpreter searches for possible rules to apply in the order they appear in the program, and there is also a deterministic strategy for finding companion constraints to form a match with the entire head of a rule. This means that non-confluent and even inconsistent programs may still be both readable and preserve a reasonable semantics, although it may go beyond a standard first-order semantics. This provides a style of programming in which simplifications and simpagations are explicitly used for their effect of deleting or revising constraints in the store. A formal semantics for CHR based on linear logic has been suggested by [5] to cope with such programs. In practice, we do not need such apparatus, and in the examples below we can specify a non-standard, possible worlds semantics when relevant.

In the constraint solvers to be shown below, we use the Prolog facility *fail* as a way to indicate an inconsistent state. When used within a larger program, this results in no new store being generated and instead the interpreter backtracks, perhaps leading to a failure of the entire computation.

3.2 Constraint Solvers for Standard and Non-Standard Semantics

A constraint solver for a standard semantics will typically accumulate the constraints into the growing state, however, taking care to

- avoid adding constraints that are already in, or implied by the current state, and
- detect failure when it occurs.

The following example shows a pattern that can be used for a large class of standard semantics.

Example 3 (Standard semantics). The semantics of Example 1 and the solver of Example 2 can be represented in CHR as follows.

```
:- chr_constraint in/2.
in(I, L1) \ in(I, L2) <=> L1=L2.
```

The first line is necessary to inform the interpreter that the `in`/2 predicate should be treated in a special way, i.e, as a constraint. The single rule of the program, in case it applies, will ensure that no individual can be registered at two different locations at the same time. If the unification `L1=L2` fails, it indicates an inconsistency. Notice that the rule is a simpagation that removes one of the constraints, thus avoiding duplicate constraints to pile up in the store.[2] If the rule does not apply when a constraint is called, it means that the constraint is the first one referring to its particular individual, and it is simply added to the store.

Example 4 (Standard semantics). We consider a constraint solver to be used for the analysis of stories about the students at small university at some fixed moment of time. The university has a number of rooms and other places, where a student can be.

`lecture_hall_1`, `lecture_hall_2`, `reading_room`,
`student_bar`, `garden`, ...

There are two courses going on, `programming_course` in `lecture_hall_1` and `linguistics_course` in `lecture_hall_2`. We have constraints `in(`s`,`r`)` indicating that student s is in room r, `attends(`s`,`c`)` that student s attends course c, `can_see(`s_1`,`s_2`)` that student s_1 can see student s_2, and finally `reading(`s`)` indicating that student s is reading. A student can only be in one room at a time, and reading can take place in any other room than the lecture halls, and for student x to see student y, they must be in the same room. A constraint solver for this can be expressed in CHR as follows; the constraint `diff(`x`,`y`)` is a standard device indicating that x and y must be different (easily defined in CHR; left out for reasons of space).

```
:- chr_constraint attends/2, in/2, can_see/2, reading/1.
attends(St, programming_course) ==> in(St, lecture_hall_1).
attends(St, linguistics_course) ==> in(St, lecture_hall_2).
in(St, R1) \ in(St, R2) <=> R1=R2.
reading(St) ==> in(St, R),
                diff(R, lecture_hall_1),diff(R, lecture_hall_2).
can_see(St1,St2) ==> in(St1,R), in(St2,R).
```

The first line introduces the constraint predicates, and the rules describe the general world knowledge explained above and at the same time it defines the set of consistent constraint stores. As stated above, we may use ground, consistent constraint stores as possible worlds in a semantics. As an example of running this constraint solver, we observe:

`[attends(peter,linguistics_course),can_see(mary,peter)]`
$\ni$ `in(mary,lecture_hall_1).`

[2] Most implementations of CHR are based on a multiset semantics; some implementations has an option for switching to a set semantics, but for reasons of efficiency, this is discouraged. It is recommended to use relevant simpagations for duplicate elimination as shown in the example.

This constraint solver is suited for reasoning about a static world, but standard semantics can also capture development over time if world facts are equipped with time stamps or abstract time representations as in the event calculus [26].

Example 5 (Non-standard semantics). We consider a robot in a two-dimensional world whose actions are to move forward, to turn left and to turn right. We will use a constraint solver written in CHR to determine the robot's position after a sequence of actions. Constraint predicates are `position`/2, intended to hold the current x and y coordinates, `direction`/1 whose argument are expected to be one of `north`, `west`, `south`, `east`, and finally nul-ary constraints for the actions `step_forward`, `turn_left` and `turn_right`. This is defined in CHR as follows.

```
:- chr_constraint position/2, direction/1,
   step_forward/0, turn_left/0, turn_right/0.

direction(north), turn_right <=> direction(east).
7 similar rules
direction(north) \ position(X,Y), step_forward
                           <=> Y1 is Y+1, position(X,Y1).
3 similar rules
```

As normalized constraint stores, we consider in this example only ground ones (for simplicity only[3]) of the form `position`$(x,y) \wedge$ `direction`(d), where x,y are integers and d one of `north`, `east`, `south`, `west`. To define a semantics, we define $\mathscr{W}$ as the set of normalized states, and W as follows.

$$\begin{array}{rl} W(s) = \{s\} & \text{for any normalized con. store } s \\ W(s \wedge a) = \{s'\} & \text{for normalized con. store } s \text{ and action } a\text{; } s' \text{ is a copy of } s \text{ with} \\ & \text{the } \texttt{direction} \text{ or } \texttt{position} \text{ fact adjusted according to } a \\ W(s) & \text{undefined in all other cases} \end{array}$$

This is obviously a non-standard semantics as conjunction does not correspond to intersection of world sets. The CHR program above defines a constraint solver for W with $s,a \vdash W(s \wedge a)$ for normalized state s and action a, and undefined otherwise. To see that the order normalization steps do matter, consider the following with $s_0 = \{$`position(0,0),direction(north)`.

$$[s_0, \texttt{step\_forward}, \texttt{turn\_left}]$$
$$= \{\texttt{position(0,1),direction(west)}\}$$

[3] The constraint solver uses a predicate "`is`" which is a Prolog device for arithmetic that only works when all variables in its right hand side argument are given at the time of the call. Replacing it by a proper constraint solver capable of handling equations concerning the addition and subtraction of the constant one, will make it possible to work with non-ground constraints, corresponding to calculating the robot's position and direction relative to an unknown start position.

$[s_0, \texttt{turn\_left}, \texttt{step\_forward}]$
$= \{\texttt{position(-1,0)}, \texttt{direction(west)}\}$

This is a minimalist example of a dynamically developing context for which each new piece of information (here: a next constraint) results in a revision rather than an addition. The order in which the constraints are encountered is essential and defines a discrete time axis.

3.3 Nondeterministic Constraint Solvers

In some cases it may be difficult to represent accommodation of a new constraint in a single new constraint store, and instead we may have the constraint solver produce different, alternative updated stores, corresponding to a disjunction of alternative interpretations. In the setting of CHR embedded in Prolog, this may be handled by backtracking. First a formal definition.

Definition 4. Let a constraint framework $F = \langle \mathscr{C}, \mathscr{W}, W \rangle$ with a set of *normalized constraint* stores be given. A subset of normalized constraint stores S is called a *normalized form of* a store s, whenever, for any substitution σ for variables of s and S, that

$$W(s\sigma) = \bigcup_{s' \in S} W(s'\sigma). \tag{3}$$

A *nondeterministic constraint solver* is a relation, denoted $s, c \vdash s'$, between normalized constraint stores s, s' and constraint c; let $S(s,c)$ denotes the set of all s'_i with $s, c \vdash s'_i$.

It is a solver *for W* whenever $S(s,c)$ is a normalized form of $s \wedge c$. It is *sound* if $S(s,c) = \{\mathit{false}\}$ whenever $s \wedge c$ is inconsistent.

Nondeterminism may be relevant for both standard and non-standard semantics. The notation of section 2 for sequences of insertions of constraints is generalized writing "$\vee$" between alternative states. Here we show an example of a constraint framework with a standard semantics and a nondeterministic constraint solver.

Example 6 (Nondeterministic constraint solver). We modify the solver shown in example 4 by changing the rule of the form `can_see(St1,St2) ==>` ... that states consequences of the knowledge that students can see each other. The new rule is as follows, where `skypes`/2 is a new constraint indicating that two students are having a video chat.

```
can_see(St1,St2) ==> in(St1,R), in(St2,R)
    ; skypes(St1,St2), in(St1,R1), in(St2,R2), diff(R1,R2).
```

The semicolon stands for Prolog's disjunction that is implemented by backtracking. Notice that the rule incorporates a (claimed) world property, that two people will not skype together when they anyhow are in the same room. The semantics is defined as a straighforward extension of the one given in example 4. Considering the steps of this constraint solver, we may possibly have $[\ldots, can\_see(peter, mary)] = s_1 \vee s_2$, where each of s_1, s_2 represents that either Peter and Mary are in the same room or in different rooms and skyping. In case the existing constraint store indicates that both Peter and Mary in fact are in the same room, s_2 will vanish, thus $[\ldots, can\_see(peter, mary)] = s_1$.

4 Driver Algorithms

In order to use a constraint solver for automatic extraction of context information from an observed phenomenon (such as a text, etc.) it needs to be combined with an algorithm that processes the phenomenon, converting it into constraints that in turn are handled by the constraint solver.

Here we use the logic programming language Prolog that plays well together with constraint solvers written in CHR. We expect a basic familiarity with Prolog and its grammar notation, Definite Clause Grammars.

We explain first the important result that the combination of Prolog as driver and CHR for context management is provably an instance of abductive reasoning. Second, we show how the special sort of Prolog programs, dressed up as Definite Clause Grammars, works seamlessly together with constraint solvers written in CHR. For reasons of space we leave out other examples of driver algorithms written in Prolog or CHR, but discuss a few options in section 4.3 below.

4.1 A Close Relationship between Prolog with CHR and Abductive Reasoning

The term abductive reasoning that stems back to C.S. Peirce, means basically to reason for a best explanation for an observed phenomenon.

While Prolog in itself is a purely deductive paradigm, different approaches to so-called abductive logic programming (ALP) have emerged since the early 1990s, and there is a direct equivalence between Prolog programs using CHR and a class of ALP programs as demonstrated by [11]. Here we will give a brief informal background in terms of an example.

An ALP program consists of a Prolog program in which certain predicates are recognized as abducibles plus so-called integrity constraints, that are restrictions on which combinations of abducibles are allowed. An abductive answer A to a query Q to an ALP is a set of abducible atoms such that

1. if *A* is added to the program as ordinary facts, *Q* would succeed according to the traditional logic semantics for Prolog, and
2. *A* satisfies the integrity constraints.

We can illustrate this in a simplistic example of an ALP program, where we also show its equivalent program in CHR+Prolog.

ALP:	Prolog+CHR:
Abducible pred's: `a`, `b`, `c`	`:- chr_constraint a,b,c.`
Integrity const's: $\neg(\mathtt{a} \wedge \mathtt{b})$	`a, b <=> fail.`
`p:- q, a.`	`p:- q, a.`
`q:- b.`	`q:- b.`
`q:- c.`	`q:- c.`

Considering the program clauses as a plain Prolog program, the query `q` would simply fail, as the predicates `a`, `b` and `c` are false. Switching to abduction, we consider the query `q` as an observation – we have observed it and insist on it being true – and we need to figure out which yet unkown facts of abducible predicates that should be added in order to make `q` true in the program. In the example program, obviously `a,c` is the only possible extension to the program that will make it possible to prove `q` true and that does not conflict with the integrity constraints. Comparing with the program to the right, written in Prolog and CHR, we see that $\{\mathtt{a},\mathtt{c}\}$ is exactly the only final store produced for the query `q`. We can convince ourselves about the validity of that solution by manually adding the facts `a` and `c` to the program.

Prolog and manual editing:

```
p:- q, a.
q:- b.
q:- c.
a.
c.
```

Query `q` will succeed and the proof includes the newly added clauses.

4.2 *CHR together with Definite Clause Grammars for Text Analysis*

CHR works seamlessly together with Prolog's Definite Clause Grammar (DCG) notation as shown in the following example.

Example 7. We consider the nondeterministic constraint solver of example 6, and add to the code, the following grammar rules as driver algorithm.

```
story --> [] ; s, ['.'], story.
s --> np(St1),  [sees],  np(St2), {can_see(St1,St2)}.
s --> np(St),   [is,at], np(C), {attends(St,C)}.
s --> np(St), [is,reading], {reading(St)}.
np(peter)     --> [peter].
np(mary)      --> [mary].
np(jane)      --> [jane].
np(programming_course)  --> [the,programming,course].
np(linguistics_course) --> [the,linguistics,course].
```

Traditionally in DCGs, the code inside the curly brackets is used for calculating attributes associated with the grammar symbols, but here we use them also for mentioning those contextual facts that must be a premise for the indicated sentences to be correctly uttered.

These facts are not known in advance, but are abduced on the flight when needed, which is the same as adding them to the constraint store. Consider the query `phrase(story,[peter,···])`, where the list represents the text *Peter sees Mary. Peter sees Jane. Peter is at the programming course. Mary is at the programming course. Jane is reading.* The resulting constraint store – i.e., the context representation for this text – consists of the following constraints. The variable written as "`X`" stands for Jane's location which we do not know much about, except that it is not one of the lecture halls.

```
attends(mary,programming_course)      can_see(peter,jane)
attends(peter,programming_course)     can_see(peter,mary)
in(jane,X)                            reading(jane)
in(mary,lecture_hall_1)               skypes(peter,jane)
in(peter,lecture_hall_1)              diff(X,lecture_hall_2)
                                      diff(lecture_hall_1,X)
```

4.3 Refinements of Driver Algorithms in Prolog, DCG and CHR

The use of Prolog with CHR allows also for having the driver algorithm to inspect the current constraint store which gives a high flexibilty for control. In [15], pronoun resolution has been approached by adding to the constraint store information about position in the text and various attributes (such as gender etc.) for the possible entities that may be referred to by pronouns.

In the so-called CHR Grammars [8], syntactic parsing is taken care of by CHR rules compiled from a high-level grammar notation with very powerful, context-sensitive rules that also interact with abduction. This means that the constraint store integrates the contextual knowledge base with the grammatical symbols.

The methods we have described extend easily to systems with multiple and perhaps partly shared knowledge bases for different agents' beliefs. Each agent is given an index that is included as an additional arguments to the constraints belonging to its knowledge base. For example, the fact `a(k)` for agent `7`, is made into `a(7,k)`; a general rule `a(X)==>b(X)` must be written as `a(Agent,X)==>b(Agent,X)` and if it is specific for agent `7` as `a(7,X)==>b(7,X)`. This should make it possible to model dialogues with exchange of knowledge, but this has not been tested systematically yet.

In [12], a different approach is suggested for use in interactive installations with several concurrent processes and perhaps streaming sensor data. Here each process has its own program, and selected constraint predicates reside in shared files, which thus also serve as communication channels.

5 Related Work and Background

The relationship between constraint logic programming and abductive reasoning was observed in an early paper [27] from 1992, before CHR was introduced in 1992-3. While the recognition of CHR as suitable for general knowledge representation and reasoning emerged through the following decade, its close relationship to abductive logic programming with integrity constraints was first reported in 2000 [1]. The use of CHR's abductive capabilities for context comprehension was suggested together with CHR Grammars in 2002 [8]. The combination of Prolog and CHR for abduction, including with DCG as featured in the present book chapter, was unfolded in the HYPROLOG framework [13] in 2005. Its relation to abductive logic programming was formally characterized and proved in [11]. Non-monotonic (i.e., non-standard) semantics[4] and its implementation in CHR were introduced in 2006 by [9] for an implementation of a paradigm called global abduction; similar uses of CHR for knowledge base update is also described by [20]. Probabilistic versions of abduction with CHR and Prolog are introduced by [10, 16]; here each abductive answer is given a probability and the most probable answer is taken as the best one.

A paper from 2005 [14] argues that the use of CHR for context comprehension leads to an integration of the traditionally separated levels of semantics and pragmatics analyses (the latter here referring to the mapping of semantic placeholders to indexes of real world entities), motivating a suggestion for a "pragmatic semantics".

The principle of language interpretation as abduction was first formulated [24] in 1993, and was inherent in earlier and parallel work, e.g., [7]. The paper [24]

[4] Abductive reasoning is often mentioned as a special case of non-monotonicity since conclusions are drawn that may not be a logical consequence of the present knowledge base. However, what we call standard semantics used in relation to abduction is a first-order, monotonic semantics for the constraint stores (knowledge bases) with a knowledge assimilation mechanism that conforms with conjunction. Readers puzzled by this discussion may find the paper [17] from 1991 interesting, in which the relation between abduction and deduction is investigated in a way that has strong links to the work presented here.

introduced also a system for discourse analysis involving a weighting scheme (analogous to, but not the same as, probabilities, cf. above). There is also a clear relation between our work and the flat representation suggested by [23].

Our non-standard semantics is related to the work on so-called belief revision, e.g., [2, 21, 22], in which general heuristics are considered on how to assimilate new observations that conflict with the current knowledge base. The main difference is that with CHR, the developer can define his or her own ways to revise the knowledge base.

In addition to the line of work described here, there is a long tradition for abduction in logic programming with and without constraints, reviewed by [18], but the identification of abducibles with constraints as in the present work is not made. A detailed comparison shows that the CHR based approach described here is likely the most efficient implementation of abductive logic programming, and the price to be paid for this is a limited support of negation.

6 Conclusion

We have exposed the similarity between context representation and a logically based approach to constraint programming, namely Constraint Handling Rules. It has been shown that this gives rise to practical methods for discourse analysis with context comprehension, and we demonstrated that this can be understood as an instance of "Interpretation as Abduction". The main advantage of the approach is the ease with which a model of the relation between a language syntax and its contextual meaning can be specified, and it can be practiced using standard, implemented tools such as Prolog, Definite Clause Grammars and Constraint Handling Rules. We showed this here only with simplistic syntactic analysis, but it should be kept in mind that we can rely on the large body of experience on using logic programming for syntax analysis and compositional (context-independent) semantics; see, e.g., [28] or the vast amount of more recent textbooks on this matter.

When used in teaching, the approach indicates a steep learning curve, as has been verified at a number of summer schools and tutorials; this goes for both audiences of linguist and of computer science students. It is well-known for decades that students can learn to write simple Prolog programs and language parsers in a few hours, and extending with CHR does not present any special obstacle – when introduced with language or simple reasoning tasks that everyone can relate to. Interestingly, our mechanisms are essentially abduction and abductive interpretation but presented in this ways, they appear quite natural even for novice students; these subjects are normally considered highly advanced and difficult.

Concerned with efficiency, our methods may give rise to both very slow and very fast language analyzers. As is well known, parsers written in Prolog can easily run into combinatorial explosions due to badly controlled backtracking, but experienced grammar writers (i.e., logic programmers) know how to control that. The addition of a contextual module in the shape of a CHR program may not slow down execution in a noticeable way, as 1) the available implementations of CHR are efficient and fully integrated with the Prolog engine, and 2) the actual CHR rules used in our examples are in most cases quite straightforward and do no give rise to any deep levels of recursion. However, as we have shown in section 3.3, it may sometimes be relevant to use disjunction within the body of a CHR rule, represented by Prolog's semicolon operator implemented by backtracking. Used badly, this may also lead to combinatorial explosions. This can to some extent be remedied by additional atomic constraints, that represent the disjunction of other constraints; however, such attempts tend to involve a huge set of strange CHR rules only to deal with disjunctions. This destroys the elegancy and transparency of using CHR for context comprehension and is the reason, why we did not feature this option.

As we have shown, the most obvious and intuitive applications of our approach concern analysis of text describing static sceneries: bits of knowledge are added incrementally, where at each point in the text a few CHR rules may be used to integrate the new knowledge and perhaps using the already found information for disambiguation. To integrate aspects of a dynamic evolution – not only in the acquisition of knowledge, but when the described state of affairs is changing over time – we showed how CHR can represent non-standard semantics, which allow effective knowledge updates, so that the constraint store at any point represents a current "now" knowledge database.

Our approach has been used by [15] to build UML class diagrams from text describing use cases, including with pronoun resolution and other linguistic refinements. The paper [6] concentrates on using CHR to resolve relative time expression appearing in bibliographical text.

The main practical limitation of the approach is that there is no standard integration with tools such as POS taggers (for the input side) or external knowledge representation systems such as a database (for the output of a discourse analysis). Fragments of such interfaces exist in different applications, but a ready to use environment with these facilities is still lacking. Thus, to use Prolog and CHR for context comprehension in a large-scale practical application, the concise programs we have shown here need to be complemented with a certain amount of detailed interface programming.

References

1. Abdennadher, S., Christiansen, H.: An experimental CLP platform for integrity constraints and abduction. In: Proceedings of FQAS2000, Flexible Query Answering Systems: Advances in Soft Computing series, pp. 141–152. Physica-Verlag (Springer) (2000)
2. Alchourrón, C.E., Gärdenfors, P., Makinson, D.: On the logic of theory change: Partial meet contraction and revision functions. Journal of Symbolic Logic **50**(2), 510–530 (1985)
3. Aliseda, A.: Abductive Reasoning: Logical Investigations into Discovery and Explanation. Synthese library. Springer (2006)
4. Apt, K.: Principles of Constraint Programming. Cambridge University Press (2003)

5. Betz, H., Frühwirth, T.W.: A linear-logic semantics for Constraint Handling Rules. In: P. van Beek (ed.) Constraint Programming, *Lecture Notes in Computer Science*, vol. 3709, pp. 137–151. Springer (2005)
6. van de Camp, M., Christiansen, H.: Resolving relative time expressions in Dutch text with Constraint Handling Rules. In: D. Duchier, Y. Parmentier (eds.) CSLP, *Lecture Notes in Computer Science*, vol. 8114, pp. 166–177. Springer (2012)
7. Charniak, E., McDermott, D.: Introduction to Artificial Intelligence. Addison-Wesley Publishing Company (1985)
8. Christiansen, H.: CHR Grammars. Int'l Journal on Theory and Practice of Logic Programming **5**(4-5), 467–501 (2005)
9. Christiansen, H.: On the implementation of global abduction. In: K. Inoue, K. Satoh, F. Toni (eds.) CLIMA VII, *Lecture Notes in Computer Science*, vol. 4371, pp. 226–245. Springer (2006)
10. Christiansen, H.: Implementing probabilistic abductive logic programming with Constraint Handling Rules. In: T. Schrijvers, T.W. Frühwirth (eds.) Constraint Handling Rules, *Lecture Notes in Computer Science*, vol. 5388, pp. 85–118. Springer (2008)
11. Christiansen, H.: Executable specifications for hypothesis-based reasoning with Prolog and Constraint Handling Rules. J. Applied Logic **7**(3), 341–362 (2009)
12. Christiansen, H.: An adaptation of Constraint Handling Rules for interactive and intelligent installations. In: J. Sneyers, T.W. Frühwirth (eds.) CHR '12: Proc. 9th Workshop on Constraint Handling Rules, pp. 1–15. K.U.Leuven, Department of Computer Science, CW 624 (2012)
13. Christiansen, H., Dahl, V.: HYPROLOG: A new logic programming language with assumptions and abduction. In: M. Gabbrielli, G. Gupta (eds.) ICLP, *Lecture Notes in Computer Science*, vol. 3668, pp. 159–173. Springer (2005)
14. Christiansen, H., Dahl, V.: Meaning in Context. In: A. Dey, B. Kokinov, D. Leake, R. Turner (eds.) Proceedings of Fifth International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT-05), *Lecture Notes in Artificial Intelligence*, vol. 3554, pp. 97–111 (2005)
15. Christiansen, H., Have, C.T., Tveitane, K.: From use cases to UML class diagrams using logic grammars and constraints. In: RANLP '07: Proc. Intl. Conf. Recent Adv. Nat. Lang. Processing, pp. 128–132 (2007)
16. Christiansen, H., Saleh, A.H.: Modeling dependent events with CHRiSM for probabilistic abduction. In: J. Sneyers (ed.) CHR '11: Proc. 8th Workshop on Constraint Handling Rules, pp. 48–63. GUC, Technical report (2011)
17. Console, L., Dupré, D.T., Torasso, P.: On the relationship between abduction and deduction. Journal of Logic and Computation **1**(5), 661–690 (1991)
18. Denecker, M., Kakas, A.C.: Abduction in logic programming. In: A.C. Kakas, F. Sadri (eds.) Computational Logic: Logic Programming and Beyond, *Lecture Notes in Computer Science*, vol. 2407, pp. 402–436. Springer (2002)
19. Frühwirth, T.W.: Theory and practice of Constraint Handling Rules. Journal of Logic Programming **37**(1-3), 95–138 (1998)
20. Frühwirth, T.W.: Constraint Handling Rules. Cambridge University Press (2009)
21. Gärdenfors, P.: Belief revision and nonmonotonic logic: Two sides of the same coin? In: ECAI, pp. 768–773 (1990)
22. Gärdenfors, P., Rott, H.: Belief revision. In: D.M. Gabbay, C.J. Hogger, J.A. Robinson (eds.) Handbook of Logic in Artificial Intelligence and Logic Programming, *Epistemic and Temporal Reasoning*, vol. IV, pp. 35–132. Oxford University Press (1995)
23. Hobbs, J.R.: Ontological promiscuity. In: ACL, 23rd Annual Meeting of the Association for Computational Linguistics, 8-12 July 1985, University of Chicago, Chicago, Illinois, USA, Proceedings, pp. 61–69. ACL (1985)
24. Hobbs, J.R., Stickel, M.E., Appelt, D.E., Martin, P.A.: Interpretation as abduction. Artificial Intelligence **63**(1-2), 69–142 (1993)
25. Jaffar, J., Lassez, J.L.: Constraint logic programming. In: POPL, Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987, pp. 111–119 (1987)

26. Kowalski, R.A., Bowen, K.A. (eds.): Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes). MIT Press (1988)
27. Maim, E.: Abduction and constraint logic programming. In: ECAI, pp. 149–153 (1992)
28. Pereira, F.C.N., Shieber, S.M.: Prolog and Natural-Language Analysis, *CSLI Lecture Notes Series*, vol. 10. Center for the Study of Language and Information (1987)